Are Software Engineers True Engineers?

Claus Lewerentz, Heinrich Rust*

Lehrstuhl Software-Systemtechnik, BTU Cottbus, Internal Report I-12/1998

1 Introduction

Are software engineers engineers? Or more exactly: Can software engineers legitimately be called engineers? This is an ambiguous question. Its interpretation depends on the situation in which it is asked. It might be asked in a situation in which practice-oriented engineering disciplines have more chances to be sponsored publicly: In this case it would be the question if software engineering should be allowed to profit from the prestige of engineering. Or it might be asked in an academic institution: Here it might be a question leading to decisions about curriculum organisation. Or it might be asked in the context of the philosophy of sciences: Here it might imply the question if the methods applied in and types of knowledge collected in software engineering are of the same type as in accepted engineering disciplines.

In this essay, we investigate the question with the intention of identifying possibilities of fruitful adoption of engineering methods in software engineering. Thus, we rephrase our question less ambiguously as: With respect to which attitudes and professional self-understanding should software engineers strive to behave like traditional engineers?

This question, too, contains an ambiguity, because it is not clear what should be understood as "engineering". We learned from structuralists that the search for neighbouring concepts and for the differences to these can help to find critical properties of a given concept, so this is the approach we use in this essay. Our essay contains answers to questions like: How do scientists differ from engineers? Is it the same for social scientists and for natural scientists? In which respect and to which extent is an engineer an artist, and in what respect is she or he not? Which properties have engineers in common with craftsmen, and which not? Are there connections with people doing handicraft?

The concept "engineer" and the neighbouring concepts are idealized personality types which we use to structure a not too well structured field. Unfortunately, a typical real person is never a pure type. Each of us has a mix of personality traits which might be considered as typical for different types. Thus, what we deal with here necessarily is a set of social stereotypes, and any real engineer will have traits which are commonly attributed to other social stereotypes.

Views which we generated from social stereotypes are the following:

- Engineers apply theories which natural scientists and mathematicians have developed for the construction of artifacts.
- Engineers use systematic development processes.
- Quality assessments based on quantitative measurements play a dominant role in engineering.

We have to admit that our approach carries with it a problem: We can not expect that the chosen stereotypic views are systematic and conflict-free, because social stereotypes do not necessarily consist of traits forming such a consistent whole. Another problem is that stereotypes are often characterized by just a few dominant traits.

Because of the latter reason, we added some charactistics from specific engineering disciplines. In order not to have to use mere stereotypes of the engineering disciplines, we look for objective features of organizations of engineering disciplines which are especially relevant for software development.

Views collected from engineering profession organizations are:

* Reachable at: BTU, Postfach 101344, D-03013 Cottbus, Germany; Tel. +49(355)69-3881, Fax.:-3810; (lewerentz|rust)@informatik.tu-cottbus.de. Submitted for publication. Version of January 6, 1999.

– Engineering professions deal with quality improvement via reflexion.
– Engineering professions include subdisciplines dealing with technological aspects if safety, reliability and the like.
– Engineering professions show interest in development and use of standards.

There is a final perspective which we use to characterize engineers: This is a vision for capabilities for engineers which are necessary in the future. There is a lively discussion concerning such questions (compare e.g. [HS96,NP97]). Two main points can be identified:

– Future engineers will have to acquire social competences.
– Future engineers will have to learn how to deal with consequences of their work which can not be described in technical terms.

For each of the views collected in these ways, we look if software engineering fits them, and in a conclusion, we present our views regarding the question in what ways software engineering should strive to become more like engineering. Finally, we describe why we differentiate between specifics and essentials of a discipline and in what respect this difference is important for the picture of software engineers as engineers.

2 Engineering is the application of theories from the natural sciences and mathematics for construction of artifacts

A common view of engineering in general is based on the assumption that engineering disciplines are largely dependent on laws of the natural sciences. They use theories which have been developed in these sciences and in mathematics to develop apparatus to solve technical problems.

This view of engineering can be based on the common conception that starting with the age of rationalism, and culminating for the first time in the scientific revolution of the 19th century, the systematic use of scientific theories for economical goals became more widespread. Traditional work procedures were replaced by planned and evaluated innovations which were supported by machines. New scientific theories were systematically inspected for their practical applicability. Artifical apparatus became more important in production processes, in transport, and in warfare.

This view of engineering can be characterized via two differences:

– This view of engineering sees the engineers as connected to the scientists whose theories the engineers use. One characteristic difference of engineers is their relation as theory users to the natural scientists and mathematicians as theory suppliers.
– Another defining difference is the kind of theory which engineers apply. Can the practical application of theories developed in some social science, or in the humanities, also be called engineering? Normally, this is not considered to be the case.
– Finally we notice that engineers are expected to produce artifacts. This differentiates engineering disciplines from disciplines like medicine.

Thus, in this view engineering can be defined to be the **application** of theories from the **natural sciences and mathematics** for the **construction of artifacts**.

This view of engineering can be questioned. Even if the use of scientific theories seems to be an important aspect of the work of engineers, it must not necessarily be seen as the essential, defining property.

The use of scientific theories is not essential in software engineering

The first view of engineering in general sees its essence in the application of scientific theories with practical goals. To decide if software engineering is an engineering discipline in this sense, we have to check if this application of scientific theories is essential in software engineering.

The theme “practical application of theoretical scientific theories” evokes the difference between theory and practice. The scientist is seen as the theoretician whose research is guided by the quest for understanding, and the engineer is seen as the practician who wants to change the world by constructing technical systems.

To some extent, this view is plausible, but it does not show the whole picture. This has to do with the fact that software engineering is a moving discipline.

To decide if software engineering is the discipline which applies the results of computer science, computing science or informatics to practical problems, we have to investigate the concept of “informatics” and its cousins. As is normal in a quickly moving discipline, there is no commonly accepted definition for it, but there is some discussion about the correct definition. Coy [Coy97] wraps it up from a german perspective. We point out two extreme positions:

- Informatics is sometimes interpreted as the mathematical theory of information and automation. In this view, informatics forms a part of mathematics.
- Informatics is sometimes seen as the discipline that mainly reorganizes human work with the goal of making it more efficient by the use of computers. In this view, informatics would belong to the occupational sciences.

These two different definitions of informatics make explicit the difficulty of defining the discipline:

- The discipline has different fields: There are parts of informatics which are adequately described by the first definition, and there are other parts which fit the second definition better.
- The discipline is changing: The technical and mathematical facts seem to have been very dominant in the past, but in the last decade, the importance of other kinds of knowledge has risen. Of course, also today there are sub-disciplines which are dominated by the application of mathematical knowledge, like semantics of programming languages, compiler construction and model checking techniques. On the other hand, especially the success of the personal computer makes it important for the software engineer writing application programs for this kind of machine to understand the work processes which are to be supported. And it is not implausible to expect that this kind of software engineering work will grow quicker in the next years than the mathematics based work.
- On different abstraction levels, different kinds of knowledge become relevant: The goals of a software engineering project can be formulated at different system levels. An efficient algorithm for a given problem can be necessary because of the interactive nature of the task: A user which has to wait a long time for an answer from the system might find this unacceptable. In this case, the user requirement might be “acceptable system response time”, which is is a psychological concept. Somebody has to quantify this, and after that, one can check how the requirement can be met by using the right hardware, the right database structure, and the right algorithms, and all this under given system constraints.

The changing nature of the field makes it difficult for us to define an essential theory as a basis for software engineering. At least, we would not like to define one specific science as the theoretical base of software engineering. The latter has many colours.

But: Software engineering is an engineering discipline in the sense that the practical constructive application of collected knowledge is central. Research is done not just for understanding an aspect of the world, but for changing it. Thus, if we stress the term “application and construction” in the definition for engineering given above, we may already see software engineering as engineering.

It is another question if software engineering should strive to develop its own theories. We think that this is the case. Software engineering does not just apply theories developed in computer science, but it belongs itself to computer science, and it develops itself theories. Software engineers apply implicit theories about what are better or worse methods in their working field. These implicit theories encompass causal relationships and correlation between the amount of work spent for different tasks and the outcome in the form of product quality and quantity. It belongs to the tasks of academic software engineering to make these implicit theories explicit, to bring them into a form which is falsifiable, to look for possible falsifications, and to develop better theories, if this is seems necessary.

Thus, we think there is a field of theories which genuinely belongs to software engineering and not to other branches of computer science. From this it follows that we do not believe that software engineering should strive to become like the one-sided picture of general engineering we described above: Software engineering can not refrain from developing theories concerning the efficiency of different methods for software development. Much to the contrary, software engineering has to strive for dealing more consciously with both implicit and explicit theories, and it has to develop methods for their falsification, validation and comparison.

3 Use of systematic development processes is essential for engineering

One view of engineering sees its essence in systematic development processes. Of course, the industrial practice does not in all cases live up to this ideal. Thus, it is not so much the industrial reality, but it is the strive for an ideal which defines engineering disciplines in this perspective.

What are systematic development processes? One central concept here is “systematic”. A systematic development process consists of a fixed set of clearly defined activities which are executed in some coordinated manner. One important coordination is with respect to time: Some activities have to be completed before others may start. Another important coordination is with respect to cooperation: Some activities have to be executed by several people, with clearly specified tasks for each participant. The processes are systematic in the sense that the pattern of activity execution is defined in some way.

“Development” is another important concept here because the goal of the process is some product which, in the beginning, is only conceived of, and which after execution of the process exists in reality.

This view of engineering contrasts the engineer with a genius artist. An artist may also produce something, but the essence of the artistic process leading to the product can not be fixed in clearly defined activities. Today, this difference might seem odd since the essence of being an artist is not seen any more in being a genius. It might seem even more odd if we look at the etymology of “engineer”, which can be traced back to “ingenium”, which is related to “genius”. But odd as it might seem, this view considers the essence of engineering to be the codification of work processes in such a way that as few ingeniuity as possible is necessary to reach some goal.

The essence of being an artist in the oldfashioned sense is having an ingenious, non-analyzable intuition and being able to express the intuitively seen things in some miraculous way. The engineer, in contrast, follows a systematic development process and is able to express clearly the reasons for the decisions during this process. Thus, transparency to colleagues is an important feature of the process enacted by an engineer.

Does this picture of engineering tell the truth? This is probably not everywhere the case. Is it an accepted ideal in engineering professions?

Software engineering processes

In this section, we are going to investigate the question if software engineering is an engineering discipline in the sense that it consists of the execution of systematic development processes. We have to look if the different kinds of activities of a group of software engineers are coordinated in some systematic way.

Systematic processes in software engineering have recently been made quite popular by the Software Engineering Institute and associated people [PCCW93,PWG+93,Hum95]. These processes consist in the identification of clearly defined activities during software development, of clearly defined roles of the persons involved in these activities, and in the orderly combination of these activities into explicitly defined software development processes. Paulk et al. define in their Capability Maturity Model (CMM) introduction [PCCW93]:

> A *software process* can be defined as a set of activities, methods, practices, and transformations that people use to develop and maintain software and the associated products [...]. As an organization matures, the software process becomes better defined and more consistently implemented throughout the organisation.

Looked at from the CMM standpoint, software engineering is an engineering discipline, or at least: it should strive to become one, in the sense investigated here. The CMM is proposed as the conceptual base for in-

creasing the capabilities for software development in a controlled, plannable manner. It culminates in the installation of quantitative measurements of effort and product quality. The measured quantities are to be used for the comparison of process variants and their systematic optimization.

The sketched "process movement" is a big chance for software engineering. With the process concept, in combination with quantifications of spent resources and product quality, it is possible to make different approaches for solving a given problem in a comparable manner, to develop more abstract theories about well or badly working methods, and to optimize the processes with respect to the idiosyncrasies of the individuals which are using them. The process approach to software engineering can deliver a framework for the theories specific for software engineering which we mentioned in the previous chapters.

The process approach so software engineering is no panacea. Its very essence presents a major difficulty for the introduction of systematic development processes in a software developing organization: It is a threat for the autonomy of software developers. One typical strategy to retain one's autonomy is creation of intransparency, and the process approach works exactly by creating transparency, ideally even with quantitative measurability of both work progress and work products. Thus, developers are expected to have a feeling of uneasiness when they are proposed to use systematic development processes in their work, while their managers, who play the other side of the game, typically find the approach very attractive.

Such phenomena belong to the field of organizational psychology (cf. [Arg92]). Practice oriented software engineers will have to deal with these difficulties, so they should be acquainted with relevant theories from this discipline.

4 Quantitative measurements and quality control is essential in engineering

Quantitative measures are used in engineering and production for quality control, and for setting quality goals. Sometimes, these quantitative measures are crutches for the expression of quality properties of products, but often quality properties lend themselves quite well to their quantitative operationalization.

In areas in which large quantities of identical items are to be produced, statistical methods can be used to control the production process. Differing production methods then can be compared according to the expense per piece and according to product quality. Often, both products and processes can be quantitatively characterized with respect to crucial quality properties.

Quantitative measurements and quality control in software engineering

Quantitative measurements for quality control are not very common in software engineering practice, but there is some research into these applications. Statistical quality control is a crucial feature of the 'cleanroom' approach to software engineering [MDL87]. The PSP [Hum95] includes systematic collection of quality data and their use for process evaluation. But such techniques are not accepted throughout the industry. There seem to exist problems with their adoption.

Basili and Weiss defined the Goal-Question-Metric (GQM) methodology for quantitative measurement of software engineering data [BW84]. This methodology is based on the fact that there are many quality goals possible in software engineering. Typical goals include defect freedom, reusability, and maintainability.

One measurable quantity is the number of program failures per time, or per use. This quantity depends a lot on environment conditions like the number of program uses per time or the specific kind of use of the program.

Independent from a specific kind of program use is the number of defects in a program. In this respect, it is a better quality criterion than failure rates, but on the other hand, it is more difficult to measure, because for a defect to become visible, some special environment condition might be necessary. Systematic reviews, testing, or even program proving can be used to get an estimate of the defects in an existing program.

Structural properties of a program, like lines of code or the number of uses of one module in another, are both relatively simple to measure and independent of use conditions. The drawback is the problematic validity. The

GQM methodology might help to define sensible sets of metrics for specific goals, but fixing acceptability thresholds for the measures is not easy. Tool support exists in research institutions for configurable structural measures [SL97].

Altogether, we may say that there are approaches in software engineering to systematically use quantitative data in the software development process, but that these approaches are not (yet?) widely used. We expect the use of quantitative measures to become more important in software engineering in the future.

5 Engineers deal with quality improvement via reflexion

Quantitative measures are used in the engineering disciplines to find out about the quality of the products of a (socio-)technological process. This process has been designed by one or several engineers. Often, in engineering disciplines, we can separate an engineering process from a production process, where the output of the first is a description of the second.

In software development, the engineering process and the production process are not separated. In total, this process has some of the characteristics of an engineering process: There are not large quantities of the same or similar products which are produced in a more or less repetitive way, but often the envisioned product differs considerably from previous products. On the other hand, there are characteristics of a production process: Efficiency of the process is important, because the final cost of the product is heavily influenced by the development process. For an engineer designing a traditional high-output production process, this might be less the case.

The strive for efficiency in the field of production has led to the quality movement (see, e.g., [Cro79]) in which the knowledge of production workers and engineers is systematically used to optimize processes. Several methods have been developed to improve the quality of products and the efficiency of processes.

The installation of quality circles and the like as part of the production process into the weekly work makes the process reflexive: The process contains components which help people who execute the process to think about the process itself and to change it, if this is sensible.

There are several possible reasons for the necessity of reflexivity in an engineering discipline:

- The discipline might not yet be grown up. Because of insufficient knowledge, the content of work processes can not yet be fixed, and only methodic meta-rules about the development of work processes can be given.

  In this view, reflexivity of an engineering discipline is a passing stage. After sufficient consolidation and differentiation, there will be classifications for different types of requirements, of different types of customers, of different types of development teams and of different types of organizations which allow to select a process model from the stock which only has to be tailored to the project properties which are not adequately mirrored in the classification patterns. After consolidation, reflexivity about the used processes is not necessary any more for the normal engineer; this will be a task in research institutions.
- The discipline deals with so “soft” objects and products that there is not even long-term hope for sufficient project classifications. This “softness” may come from changing customer requirements or different organizational culture.

  If this is true, reflexivity must be built into the engineering discipline on a long term basis. All engineers, or at least some in every project, must be educated to be able to reflect on the processes they use, and to optimize the processes.
- As long as technological innovations relevant for the discipline continue with high speed, we cannot hope to fix the work processes.

  In this view, reflexivity is dependent on the progress of technological developments.

We can synthesize these points in the following thesis: In an environment in which learning is necessary, i.e. in a changing environment, processes have to be reflexive to stay successfull.

Reflexivity in software engineering processes

The CMM introduced an important concept: The organization of the work processes in software engineering cannot be fixed once and for all. They have to be adapted to changing needs, and we have to allow for developer specific, for customer specific, and for application specific changes. These changes to software development processes have to be controlled, i.e. the effects have to be measured and the results have to be compared to the results of other projects. In this way, helpful changes can be differentiated from less helpful ones.

The reasons which make reflexivity necessary in traditional engineering processes are equally relevant for software engineering. We do not want to investigate the question here if the reflexive nature of software engineering is a transient or a permanent phenomenon; we believe that we could fix more process properties than we do today by classification of requirements profiles and capability profiles of the software engineers, and we believe that there will alway be enough environmental change to justify institutionalized reflexivity in software engineering processes; but the following considerations do not depend on this:

- As of today, software engineering processes should be organized as reflexively. This is because we are not able to fix software development processes for all application areas.
  A consequence is that software engineers should be educated to reflect on the process they use, to decide when to take which consequences, and to perform the decisions.
- Even if we can not fix software processes in every application field of software engineering, we might be able to do so in some more stable subfields.
  This is, in fact, one of the ideas on which the CMM is based. Here, a reference software process is defined for an organization which has to be tailored for the implementation in a specific subfield. This fixing of a reference process creates the stable subfield: It insulates the organization from the uncontrolled adoption of new methods.
- There will always be need for experts for the reflexive aspects of software engineering. Changes in technology, methodologies and organizational culture will make it necessary to optimize software development processes then and again.

The software engineering process concepts which originated at the SEI stress the institutionalization of reflexion in the software processes. The Personal Software Process (PSP, [Hum95]) includes elements in which collected data is analyzed for process evaluation and controlled change, and the definition of the uppermost level of the CMM asks controlled experiments for process improvements.

We believe that each software development process definition should definitely include elements leading to reflexivity of the process. Reasons for this are the dependencies of process efficiency on specific personality traits and the knowledge of the individuals involved, and the quick innovations with respect to tools, notations and methods which can be helpful in the software development process. Thus, a given process must be optimized for the individuals, and it should have the potential to test, evaluate and possibly incorporate new process elements.

6 Engineering disciplines have subdisciplines dealing with technological aspects of safety, reliability and the like

Many engineers design and construct artifacts which are based on the application of some theory from the natural sciences, and which can be used by other people without the engineer being present. Thus, artifacts are objectified engineering knowledge which can be applied without a knowledgeable human being present.

These artifacts might carry risks with them. A difficult question is where to place the responsibility when some mishap or some other unwanted consequence occurs. Who is responsible for long-time consequences of the use of some artifact for persons or for nature? A typical example is the consequence of artifact design for work environments. Who is responsible for illness or, even more difficult to assess, for dequalification in engineering-defined working conditions? Who is responsible for influences on society at large which are

induced by specific technologies. What can be expected from the users of an artifact? What must be expected from the engineers? In [LR87] we find a collection of views on this and closely related subjects. A common view is that engineers can not be freed from their responsibilities. [Per84] investigates catastrophic accidents. [Kle91] describes many mishaps, traditionally attributed to "human error". In these cases, engineering and management errors led to (sometimes even catastrophic) mishaps which, under the given circumstances, had to be expected to happen sometime, either because of the great complexity of the technical system, or because of the unavoidability of occasional human slips and forgetfulness.

Thus, the responsible engineer will have to use knowledge about the situation in which the artifacts will be used. This includes at least knowledge about education and training of future users, as well as general knowledge from occupational psychology about influences of technical conditions on the quality of work, from psychology of attention and from similar disciplines.

Is this view a reality in engineering disciplines? Is this an ideal in the engineering discipline? One answer to these questions, which also hints at the scope of consequences which engineers try to look at, can be given by inspecting the organizations dealing with results of engineering work. Are there certification agencies for the products of engineering work? If yes, are they effective?

This is not the place for a deeper investigation of this question. We only mention that during the industrial revolution in the 19th century accidents with steam engines led to the installment of regulation agencies, initiated both from industry and governments. Safety technology is an intensively investigated subfield of engineering (cf. [PM85] for an overview).

Responsibility for consequences in software engineering

In the engineering disciplines, safety engineering forms a subfield of its own. There are branches of computer science also emphasizing this area [Lev95]. But: Certification of software seems to be more difficult than certification in traditional engineering disciplines. Some of the well known reasons depend on the discrete nature of the state space of software based systems:

- The causality between inputs and outputs can be very complicated, and is typically nonlinear. Thus, small input changes can lead to large output changes.
- The effects of a software defect are difficult to restrict to parts of the program's functionality. This is especially true for operating system defects.
- The correctness of a software based system depends on the correctness of many system components: Application software, compiler, assembler, linker, operating system, hardware are equally relevant.
- There are no easily applied and valid quality certification measures for software systems. Quality control by inspection, e.g., can be as resource intensive as development.

These difficulties make certification of software systems a difficult task. One important strategy with respect to this question is not to rely on computers and software for safety critical functions. But this approach is not always possible. Another approach is to use a well-tested hardware base on which very small programs run which can be understood on assembler level. In this way, the inspector does not have to trust a complicated compiler.

Quantification of software reliability with statistical methods, e.g. in the form of failure rates, is problematic because the causes of reliability changes are not aging processes for which acceptable general statistical models with a physico-mathematical base exist. The causes of varying software failures rates are differences in the use conditions of the system, which are much more difficult to model statistically in a general way (cf. [Rus94, chapters 6 and 7]).

Software safety as a field on its own has not yet left its infancy, but this clearly is an area in which software engineering has to strive to become like engineering, i.e.:

- Software engineering must develop specialized methods and notations and to investigate what they can guarantee, what they can not guarantee, and which strengths and weaknesses they have in comparison with each other.

– Software engineering must devise certification methods for software intensive systems which are practically usable. These methods must describe clearly what is certified by them and what is not certified.

7 Engineers use standards to manage contingency

Engineering knowledge is supposed to be applied in a world largely determined by its history. This might be contrasted with a picture of the sciences which seem to deal with a world of eternal truths. Even if this picture must be said to be quite misleading, either as a historical fact (see [Kuh70]) or as an ideal (see [Fey75]), this picture is quite common – perhaps more so as an ideal than as a historical fact.

This misleading picture is not the self-picture of engineering disciplines. Engineers know that they have to respect most of the outcomes of historical developments in all their contingency, if their technical solutions are to be accepted. Many products of their work have to be combined with other artifacts (with contingent interfaces) for a complete solution to some problem. If the construction is to be used directly by humans (i.e., if it has some user interface), it has to respect the current consensus in ergonomics. Many decisions have to be taken which can not be based on some "eternal necessity" alone.

Engineers know about the contingencies of a substantial part of the world he works in. Thus they learn to become interested in reducing contingency consequences systematically.

One way to do so is the adoption of standards, and the development of new standards where they become helpful. Successful standards reduce the consequences of contingencies in several ways: They fix vocabularies and make communication easier this way; they reduce the number of design alternatives; they allow for the combination of indepently developed artifacts. Thus, the engineering decision space gets more structured if standards are applied.

Of course, the standards themselves are another contingency with which engineers must deal. They change with time. Engineers have to learn to estimate how stable a standard is, and they have use this estimation when they plan to develop an artifact for which a given life span is planned.

Contingency management in software engineering

There are several fields in software engineering in which standards are used with the same goal:

– Design notations like flow diagrams are standardized.
– Programming languages and their accompanying libraries are standardized.
– Communication protocols and interface methods are standardized.
– Operating systems are standardized.

These technological standards help different developers to combine their solutions with existing work which can be independently developed, and they allow them to use knowledge they have collected in one project in further projects.

A problem with software engineering standards is their quick development. Progress is so fast in this field that the problem about changing standards we mentioned is very common in software engineering, and a software engineering standard which does not change is likely to be ignored after some time. Thus, life span considerations become especially important in software engineering. How long can we assume the suppliers of a needed hardware or software component of a system to exist? What about tool support?

In a quickly changing field, contingency management can mean not to be able to trust in other organizations' longevity, and not to be able to trust in standards, if solutions have to live long enough. This might in some situations be a good reason for the NIH-syndrome ("not invented here") in computer science organizations, i.e. the wish to use only locally developed solutions to one's problems – which does not mean that we expect that most NIH-syndrome occurrences have such respectable reasons.

In addition to the technological fields, there are other areas in which standards are used in software engineering projects. One is the standardization (or meta-standardization) of software processes. ISO 9001/ISO

9000-3 are examples, as well as collections of standard process elements for the design and development of safety critical systems.

These are meta-standards: They define how a software organization should document their software process definition and their enactment, they do not fix explicitly how these processes should be defined. There is not yet enough knowledge to fix this kind of development process structures, perhaps because the field is changing too rapidly and the situative factors are not yet enough understood so that they can not yet be considered methodically, but must be left to the discretion of each software developing organisation.

8 Future engineers must acquire social competences

Today, engineers work in teams. They have to work together with other engineers and with experts from other fields, and they may have to deal with customers. This makes it plausible that an engineer should not be educated only in technological matters, but that he or she must also acquire different social competences. These include:

- Effective communication with other engineers.
- Effective communication with persons of different non-engineering backgrounds.
- Capabilities to organize one's individual as well as social work processes.
- Recognizing conflicts and dealing with them productively.
- Skills in management and leading work groups.

These needs seem plausible. It is only implausible that they did not exist in the past. Why should they be more important in the future than they have been in the past? We only hint at two reasons which are sometimes given for this: The work conditions are more quickly changing, and the engineering tasks are more complex.

Social competences of future software engineers

We would like to point out one special feature of software engineering which makes the necessity for social competences explicit: Software development processes are determined not only by technology, but to a large extent also by characteristics of the individuals, of the groups, and of the organizations which are involved. Because of this, approaches from social and occupational psychology are relevant in software engineering in several respects:

- Software increasingly plays a role as "integration glue" in many technical and nontechnical application fields.
- The organisation of work processes of software engineers belongs to software engineering. There exists a lot of knowledge about good work design in occupational psychology.
- Because the design of reflexive software development processes belongs to the tasks of software engineering, the discipline must deal with organizational learning, and must know about problems with this kind of learning.
- Even if the software process itself is not reflexive, there should be possibilities to improve a given software development process. This is seen as increasingly important [Gra97, p. 2]. Concepts from the theories of organizational learning can help also with this.
- When the individual software developer is concerned, interdependencies of motivation, quantitative measurements and feedback which have been investigated in motivational psychology can be helpful.

To sum up: We think that there are many respects in which software engineering should strive to apply good ideas from engineering disciplines, but there is also a number of problems and questions in which the social sciences can be of more help than engineering disciplines.

9 Future engineers must be able to assess non-technological consequences of their work

The view that engineers have to try out whatever seems technologically possible and interesting is not very popular any more. In the 19th century, technological optimism was more widespread. The development of the atomic bomb and ecological problems of technology are often seen as having resulted in a more differentiated picture of the values of technological progress: Increase of technological ability is not good per se.

Because of this, engineers have to use other argumentation patterns to explain why society needs them. The hypothetical demands of the society of the future are sometimes used for this. This leads to consequences concerning recommendations for the education of future engineers.

If technology is not a value in itself but only an instrument for the strive for some other values, the responsible engineer must be able to evaluate different uses of technology with respect to these underlying values. Thus, he or she must acquire a professional capability to weigh different values, and thus to judge morally. We find different kinds of non-technological values which future engineers are expected to appreciate: They belong to economics, to ecology, and to the quality of life in general.

- The engineer of the future is expected to have a better appreciation of economic constraints and needs, since in a world becoming smaller, competition seems to be expected to become more intensive. This is an argument both from the perspective of companies as from the perspective of nations.
- With ecological problems becoming more dominant in the public opinon, future engineers are expected to be able to deal more consciously with phenomena like, for example, energy consumption and waste avoidance. This is closely connected with the use of the growth of the world's population as an example of a complex task requiring technological solutions.
- Another aspect is the general quality of daily life. People have come to value technological artifacts like electrical power from wall plugs, cheap transport facilities, telecommunication technology, machinery to help with hard or dangerous work, or health technology. Engineers are expected to understand these values and compare them with others.

The same approach which helps a worker not to have to risk his health in his work can push him or his colleagues into unemployment. This is typical in a practical situation: Different values can be in conflict with each other, and to take responsible decisions, the engineer has to weigh the different values against each other. The german engineers' association (Verein Deutscher Ingenieure, VDI) has developed a guideline for technology assessment which respects this fact [Ing91].

Such requirements can only be fulfilled responsibly if engineers are educated to deal with concepts from disciplines which specialize in these non-technological consequences.

Social consequences of the work of future software engineers

With respect to non-technical consequences of their work, software engineers are in the same situation as engineers in general. Their work products are highly relevant for the economy, convenience technology and health care, and they can be expected to be of use in ecology.

In an essay about necessary non-technological capabilities of computer scientists, Kornwachs [Kor97] deals with "angewandter Informatik", i.e. applied computer science. Kornwachs criticizes that today's computer science education does not emphasize general professional abilities. Some of the questions which he cites are: How does the design of software based systems influence work organization, work content and work structure? How is social reality influenced via monopolies and diversification of software products? How does standardization of software products influence the health care system?

The responsibility for one specific non-technological consequence of software engineering has been stressed by Coy [Coy89,Coy92]: Software engineers reorganize work processes. To be able to do this responsibly, they must know at least basic concepts from occupational psychology to answer questions like the following:

– How should tasks be allocated to computers and humans?
– Is vigilance a problem in the designed working situation?
– Is the working task complete, i.e. does it include not only execution but also preparation, control, planning and coordination with others?

Questions like these are investigated in occupational psychology.

With respect to consciousness for consequences of their work which can not be described in purely technological terms, software engineers seem to be in the same situation as engineers in general: There is consensus that there is some need for them to learn systematically about such things, but as of today, this need is not yet commonly fulfilled in the curricula.

10 Outlook: Specifics and essentials

Are software engineers true engineers? Our investigation had the result that there are some respects in which they are angineers, or should at least incorporate some goals from traditional engineering disciplines, and that in others, it differs from classical engineering disciplines.

During the work on this paper, we learned that it is helpful to differentiate between the specifics of a discipline and its essentials. The specifics are the respects in which a given discipline differs from other disciplines. The essentials are the capabilities which a practician of a discipline has to acquire. These two concepts help in the formulation of an important thesis: There are specific features of engineering disciplines which help to separate the concept of an engineer from neighbouring concepts like that of a natural scientist, of an occupational psychologist, of a mathematician, of a manager, of an artist, or of a master of some craft. But it is essential that an engineer not only has abilities in specific engineering areas: He must also be able to engage in productive dialogues with experts of other disciplines. For this, it is not sufficient just to be open-minded: It is also necessary to acquire basic concepts of the disciplines which whose exponents one is going to cooperate.

The same is true for software engineers, and this is perhaps the main result of our investigation. The specifics are important, but the equally important non-specific essentials are easily overlooked. If software engineers are engineers in a classical, specific sense does not matter too much. Something else is important: There are essentials of software engineering which are better developed in engineering disciplines. And: Responsible software engineers will not hesitate to learn from other disciplines what they need to do their work competently, no matter if these other disciplines belong to classical engineering professions, or if they come from some other field of knowledge or practice.

This openness is especially needed in education. The ideals of engineering disciplines can be of value also in software engineering education; it is only important to not only emphasize stereotypic specifics, but also the other essentials.

References

[Arg92] Chris Argyris. *On Organizational Learning*. Blackwell, Oxford, 1992.

[BW84] V. Basili and D. M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, SE-10(6):728–738, November 1984.

[Coy89] Wolfgang Coy. Brauchen wir eine Theorie der Informatik? *Informatik-Spektrum*, 12(5):256–266, 1989.

[Coy92] Wolfgang Coy. Für eine Theorie der Informatik! In Wolfgang Coy, Frieder Nake, Jörg-Martin Pflüger, Arno Rolf, Jürgen Seetzen, Dirk Siefkes, and Reinhard Stransfeld, editors, *Sichtweisen der Informatik*, pages 17–32. Vieweg, Braunschweig, Wiesbaden, 1992.

[Coy97] Wolfgang Coy. Defining discipline. In Ch. Freksa, M. Jantzen, and R. Valk, editors, *Foundations of Computer Science: Potential–Theory–Cognition*. Springer-Verlag, Berlin, 1997.

[Cro79] Philip B. Crosby. *Quality Is Free: The Art of Making Quality Certain*. Mentor, New American Library, New York, 1979.

[Fey75] Paul K. Feyerabend. *Against Method: Outline of an Anarchistic Theory of Knowledge*. NLB, London, 1975.

[Gra97] Robert B. Grady. *Successful Software Process Improvement*. Prentice Hall, 1997.

[HS96] Klaus Henning and Joerg E. Staufenbiel. *Berufsplanung für Ingenieure*. Staufenbiel, Köln, [11] 1996.

[Hum95] Watt S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, Reading/Massachusetts, 1995.

[Ing91] Verein Deutscher Ingenieure. *Technikbewertung – Begriffe und Grundlagen. Erläuterungen und Hinweise zur VDI-Richtline 3780*. VDI, 1991.

[Kle91] Trevor Kletz. *An Engineer's View of Human Error*. Institution of Chemical Engineers, 1991.

[Kor97] Klaus Kornwachs. Um wirklich Informatiker zu sein, genügt es nicht, Informatiker zu sein. *Informatik Spektrum*, 20(2):79–87, 1997.

[Kuh70] Thomas Kuhn. *The Structure of Scientific Revolutions*. The Univ. of Chicago, Chicago, 1970.

[Lev95] Nancy G. Leveson. *Safeware*. Addison Wesley, Reading/Massachusetts, 1995.

[LR87] H. Lenk and G. Ropohl, editors. *Technik und Ethik*. Reclam, Stuttgart, 1987.

[MDL87] H. D. Mills, M. Dyer, and R. C. Linger. Cleanroom software engineering. *IEEE Software*, pages 19–24, September 1987.

[NP97] Wolfgang Neef and Thomas Pelz, editors. *Ingenieurinnen und Ingenieure für die Zukunft*. TU Berlin, Berlin, 1997.

[PCCW93] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber. Capability Maturity Model for software, version 1.1. Technical Report CMU/SEI-93-TR-024, Software Engineering Insitute, February 1993.

[Per84] Charles Perrow. *Normal Accidents*. Basic Books, New York, 1984.

[PM85] O. H. Peters and A. Meyna, editors. *Handbuch der Sicherheitstechnik*. Carl Hanser Verlag:München, Wien, 1985. Zwei Bände.

[PWG+93] Mark C. Paulk, Charles V. Weber, Suzanne M. Garcia, Mary Beth Chrissis, and Marilyn Busch. Key practices of the Capability Maturity Model, version 1.1. Technical Report CMU/SEI-93-TR-025, Software Engineering Institute, February 1993.

[Rus94] Heinrich Rust. *Zuverlässigkeit und Verantwortung*. Vieweg, Braunschweig, Wiesbaden, 1994.

[SL97] Frank Simon and Claus Lewerentz. Integration of an object-oriented metrics tool into SNiFF+. Technical report, BTU Cottbus, 1997. I-22/1997.